Universität Potsdam
Mathematisch-Naturwissenschaftliche Fakultät

Implementierung und Evaluation eines hardwarebeschleunigten Load-Balancers auf einer Nvidia Bluefield DPU Architektur

Sten Heimbrodt

Gliederung

- Hintergrund und Motivation
- SmartNICs Allgemein
  → Definition
  → wichtige Vertreter
- Nvidia Bluefield
  → Grundlagen
  → Generation 1 & 2
  → Generation 3
- Netzwerklastverteilung
- DOCA
  → Allgemein
  → Flow
  → Open-vSwitch
- XenoFlow – L3/L4 Lastverteilung
  → Aufbau
  → Quelltext
- Messungen und Experimente
  → Forschungsfragen
- Fazit und Beantwortung
- Ausblick

Hintergrund und Motivation

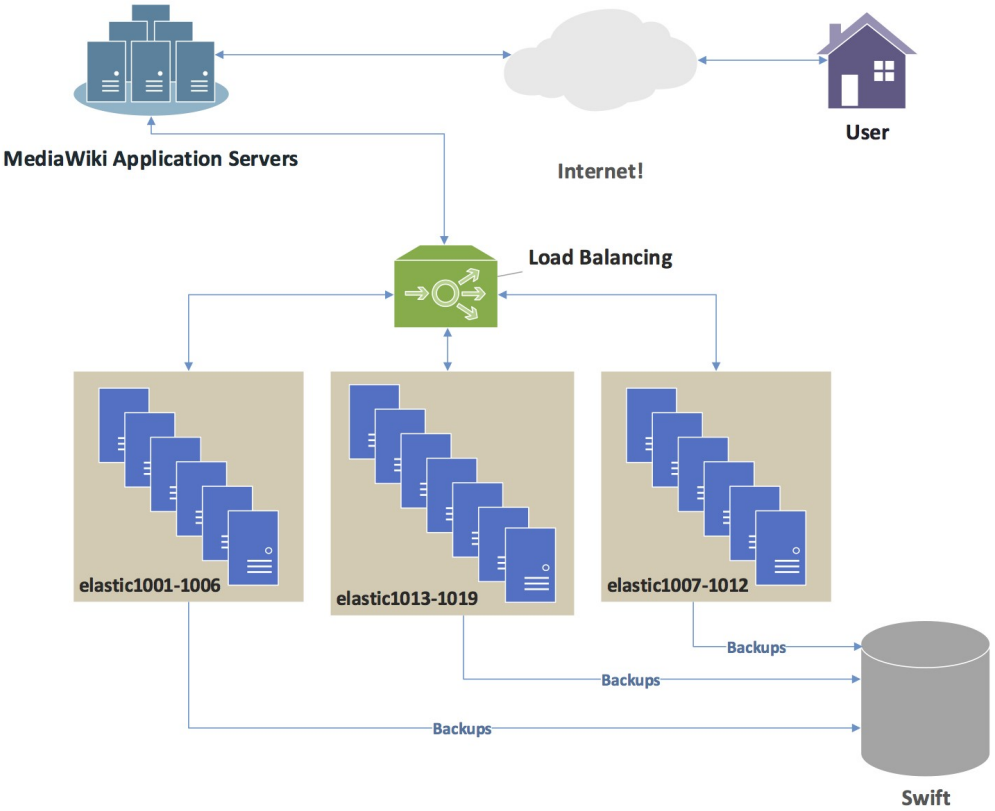
- Lastverteilung u.a. zentralstes Thema der Informatik

  → sowohl Mehrprozessor-, Mehrkern- und netzwerkverteilte Systeme

  → allerdings auch vertikale Skalierung denkbar
- Netzwerklast muss von entsprechenden Backends verarbeitet werden

  → Bewältigung der Anfragenlast aber auch Energieeffizenz
- Verteilung und Klassifizierung des Netzwerkverkehrs

  → Parameter
- Netzwerkbeschleuniger im Rahmen von dezentralen heterogenen Netzwerken geeignet



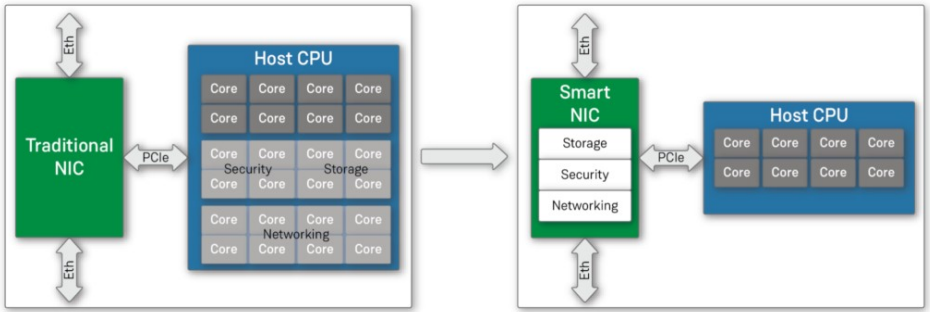
https://de.wikipedia.org/wiki/Lastverteilung_%28Informatik%29#/media/Datei:Elasticsearch_Cluster_August_2014.png

SmartNICs

- Netzwerkkarten die speziallisierte Hardware mit klassischen Systemen verbindet

  → System-on-a-Chip (SoC) mittels Schnittstellen an Prozessor angebunden

- Unscharfe Definition und schwere Abtrennung von klassischen Netzwerkkarten →



https://www.fibermall.com/de/blog/difference-between-smartnic-and-nic.htm

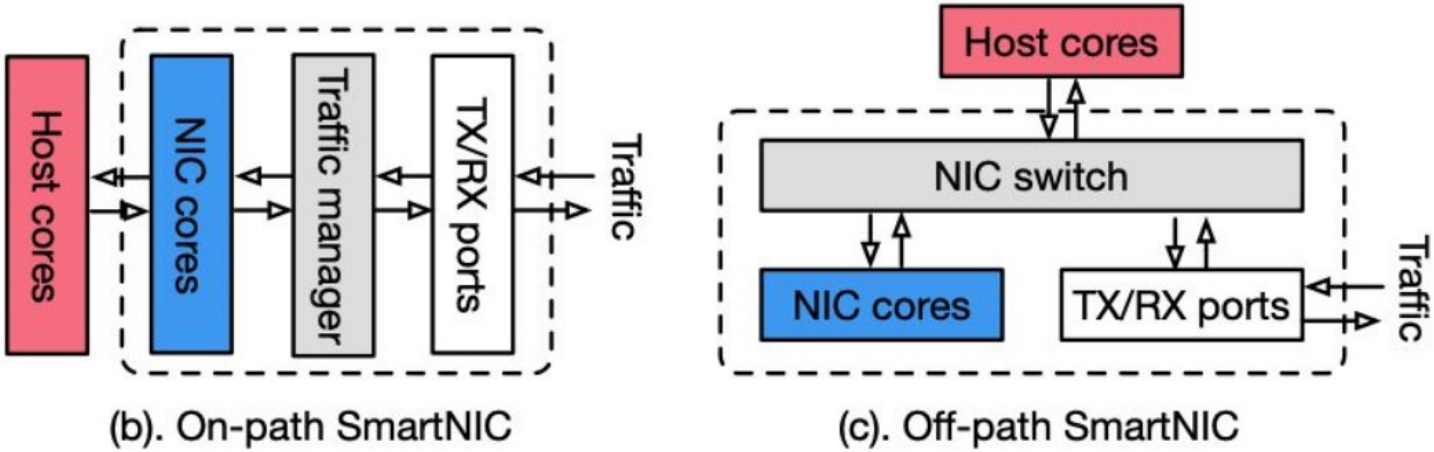
ASIC-based SmartNIC

FPGA-based SmartNIC

SoC-based SmartNIC

https://resource.fs.com/mall/generalImg/C8qTbaL28omBlQxyVf0cgOnunNb.png

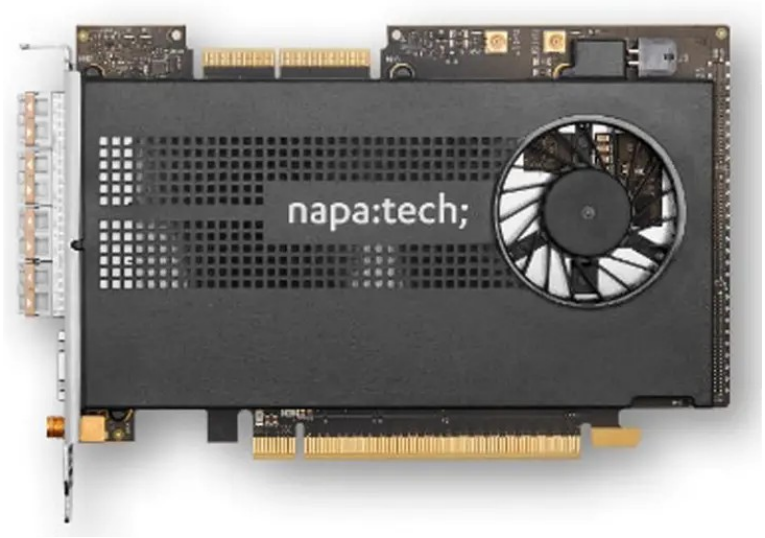
On-Path vs. Off-Path



Wichtige Vertreter

- Nvidia (Mellanox)
  - → Fokus auf AI/ML Produkte
  - → Bluefield Serie
- Broadcom
  - → eher Richtung CPU-HPC
  - → Stingray Serie
- Napatech
  - → FPGA basierte Lösungen
  - → NT100A01 und NT50B01

https://www.neox-networks.com/wp-content/uploads/Unorganisiert/NT100A01.png.webp



Napatech - NT100A01

Die Architekturen sind grundverschieden für die eigentlich selbe Funktionalität!

Nvidia Bluefield

- Bluefield kombiniert ASIC, FPGA und CPU basierte Datenverarbeitung

  → vermutlich zu lange für Marketing, daher: Data Processing Unit (DPU)
- Hintergrund war das Aufkommen heterogener Cluster

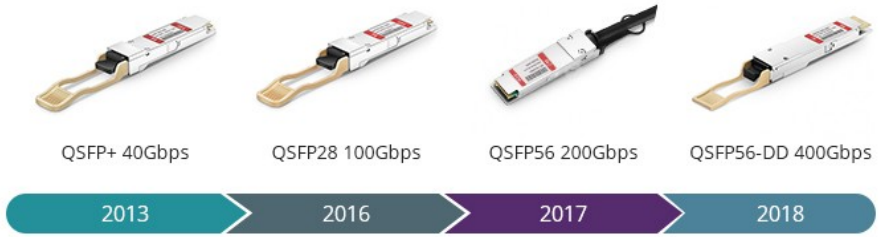
  → Aber auch anwendungsbezogen
- Bieten verschiedenste Hardwarebeschleuniger

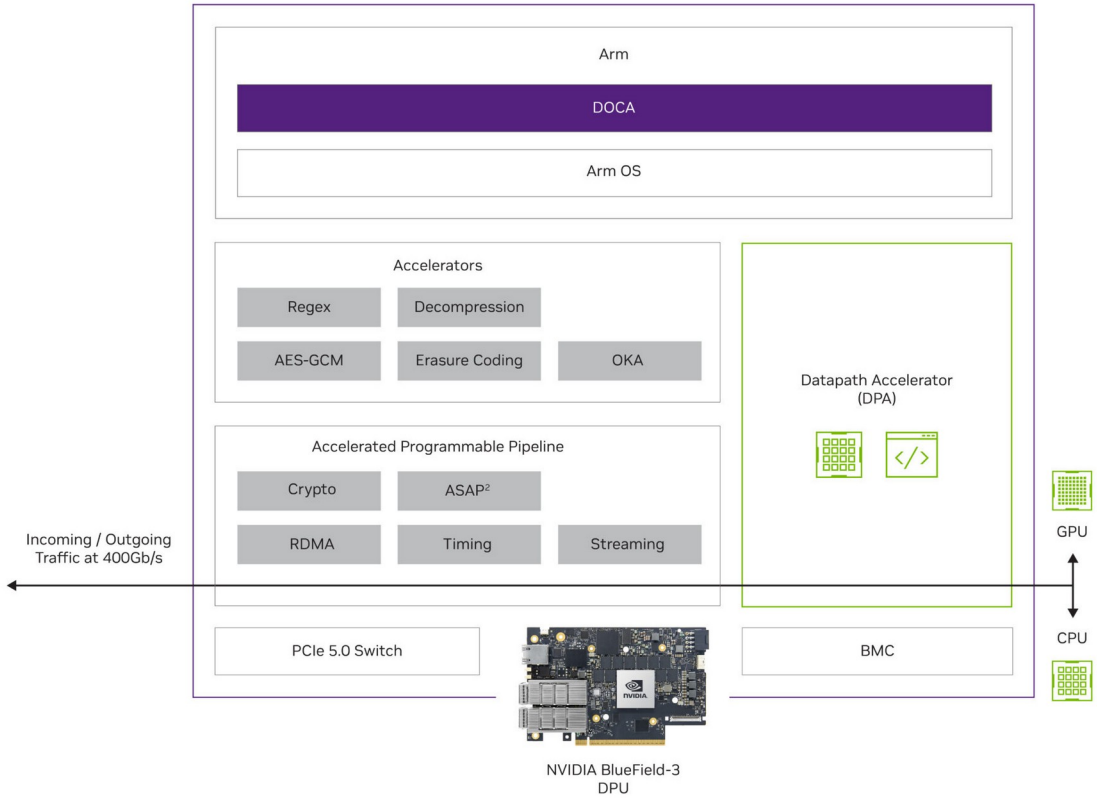
  → sollen breit im Datacenter eingesetzt werden
- Moderne Entwicklungen bieten sogar GPU Anbindung (?)

Bluefield 1 & 2

- Erste Generation Bluefield aus dem Hause Mellanox
  → damalig schon selbe DPU Architektur
  → ARM A72 + RISC-V DPA
- Ab Bluefield 2 kamen die Produkt von Nvidia
  → Übernahme von Entwicklern und Zielgruppe
- Von 100 Gbit/s auf 200 Gbit/s
  → Verdopplung des Arbeitsspeichers

https://resource.fs.com/mall/generalImg/DsAybtmm8o3kThxwGkbcIvWynHf.png





https://www.fibermall.com/de/blog/understand-nvidia-bluefield-3-dpu.htm

Bluefield 3

- Erstmalig 16x ARM A78 Kerne
- 64 GB RAM DDR5
- Option auf Bluefield-3X

  → zusätzliche Nvidia A100 auf der DPU
- PCI-E 5.0

  → 32 GT/s
- Seitens Nvidia viel breiteres Anwendungsgebiet beworben
- DPA 16 Kerne und 256(!) Threads

  → eigentlich kein direkter ASIC

Hyperscale HPC/AI

B3240 - FHHL
2P x QSFP112, 400G, >75w

HPC/AI

B3140H - HHHL
1P x QSFP112, 400G, <75w

Cloud Computing

B3210/B3220 - FHHL
2P x QSFP112, 100/200G, >75w

B3210L/B3220L - FHHL
2P x QSFP112, 100/200G, <75w

Storage

B3220SH - FHHL
2P x QSFP112, 200G, >75w

https://developer.nvidia.com/blog/power-the-next-wave-of-applications-with-nvidia-bluefield-3-dpus/

Netzwerklastverteilung

- Wird unterschieden durch Kriterium

  → Kriterium entscheidet über Klassifizierungsentscheidung

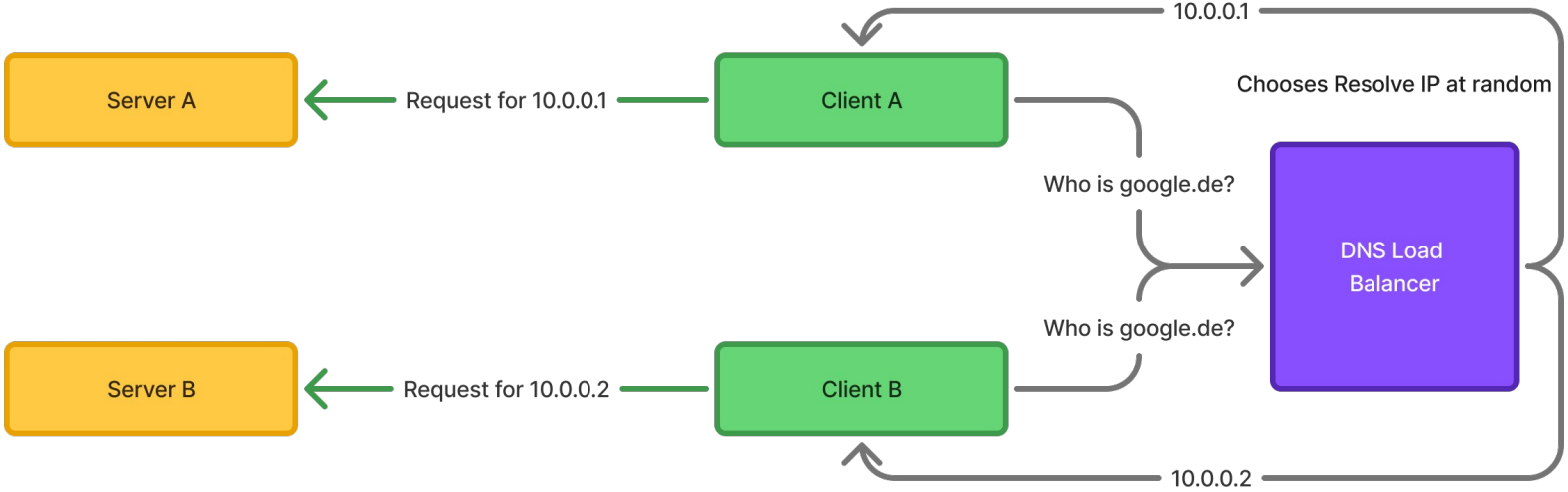
- Frühste Form war DNS-Lastverteilung

  → DNS Server gibt bei DNS Anfrage unterschiedliche IP Adresse der mehreren Backends zurück

  → fällt etwas aus der Reihe da nicht wirklich einer OSI-Schicht zuordbar

  → sehr fehleranfällig

  → kaum Konfigurierbarkeit


10.0.0.1
Server A
Request for 10.0.0.1
Client A
Chooses Resolve IP at random
Who is google.de?
DNS Load Balancer
Who is google.de?
Server B
Request for 10.0.0.2
Client B
10.0.0.2

L3/L4 Lastverteilung

- IP Header für Lastverteilung als Kriterium

  → Source/Destination IP bzw. Source/Destination Port
- Vorteile:

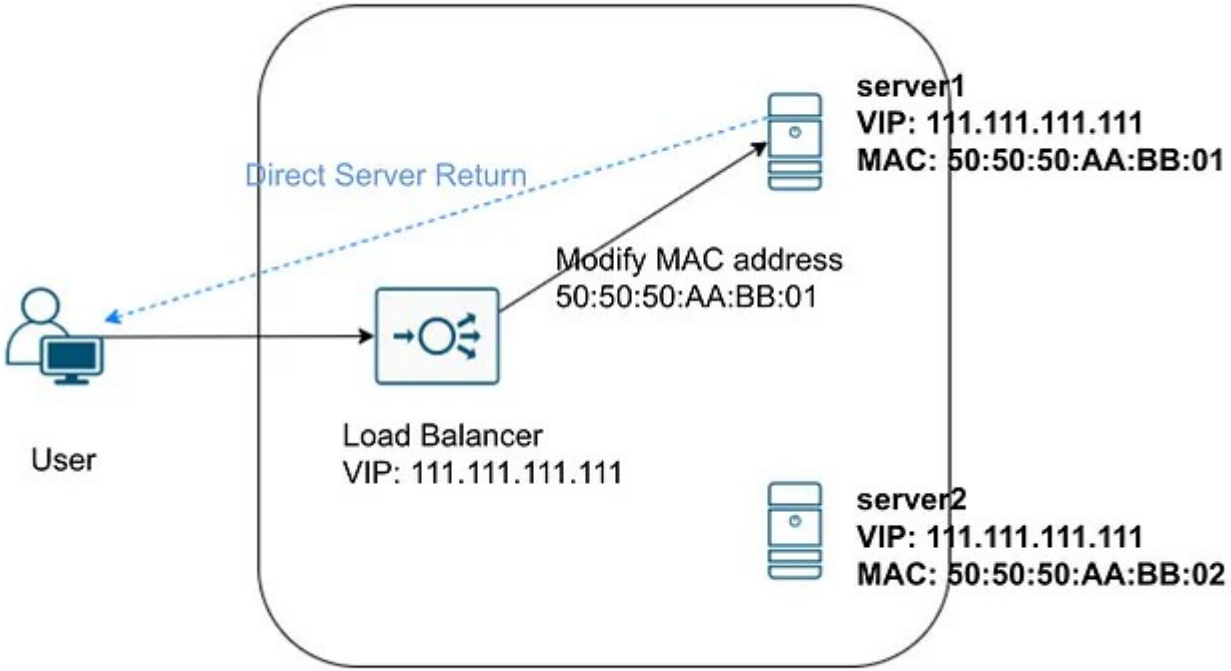
  → Wenig Overhead da kein wirkliches Parsing

  → eignet sich gut für nicht HTTP Protokolle
- Nachteile:

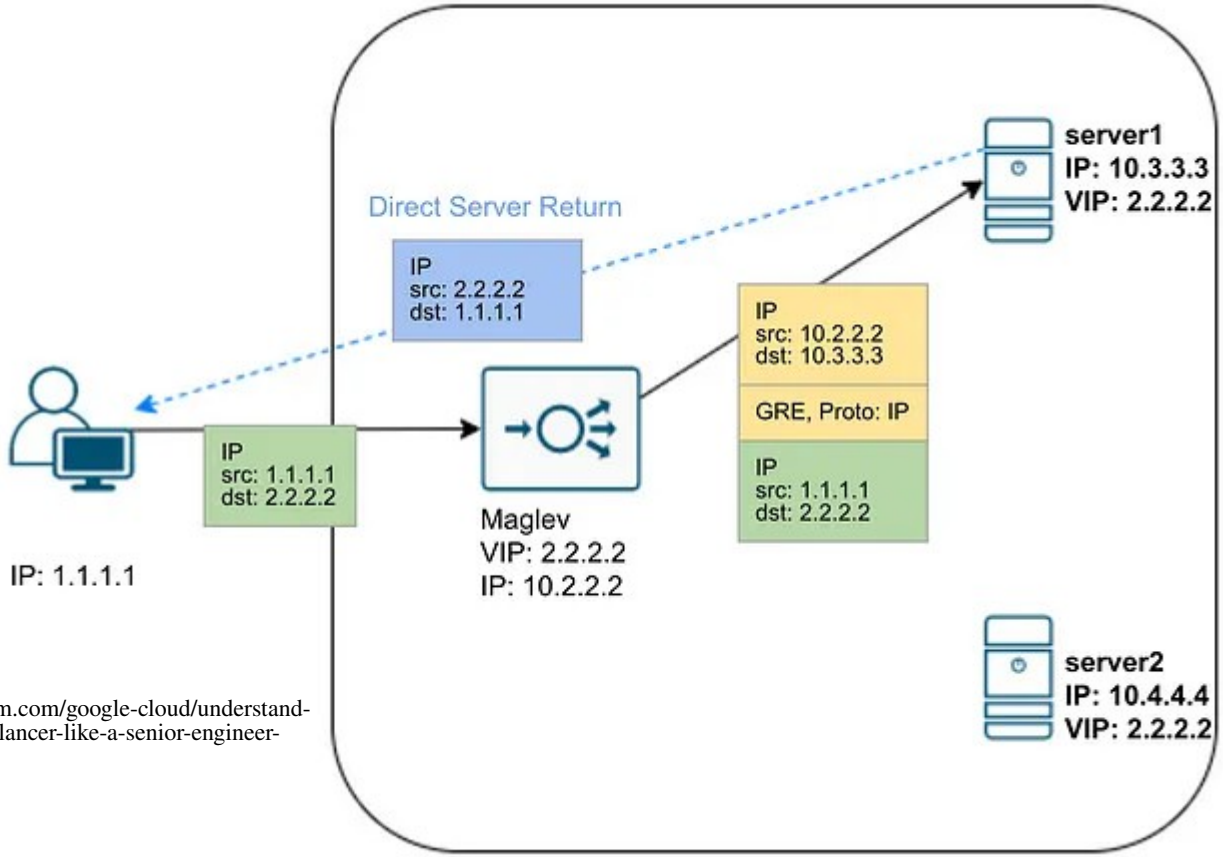
  → keine Inhaltsbasierte Lastverteilung

  → keine Cookies oder Pfade

  → wenig geeignet für Webanwendungen



https://medium.com/google-cloud/understand-cloud-load-balancer-like-a-senior-engineer-d4f55f3111fc

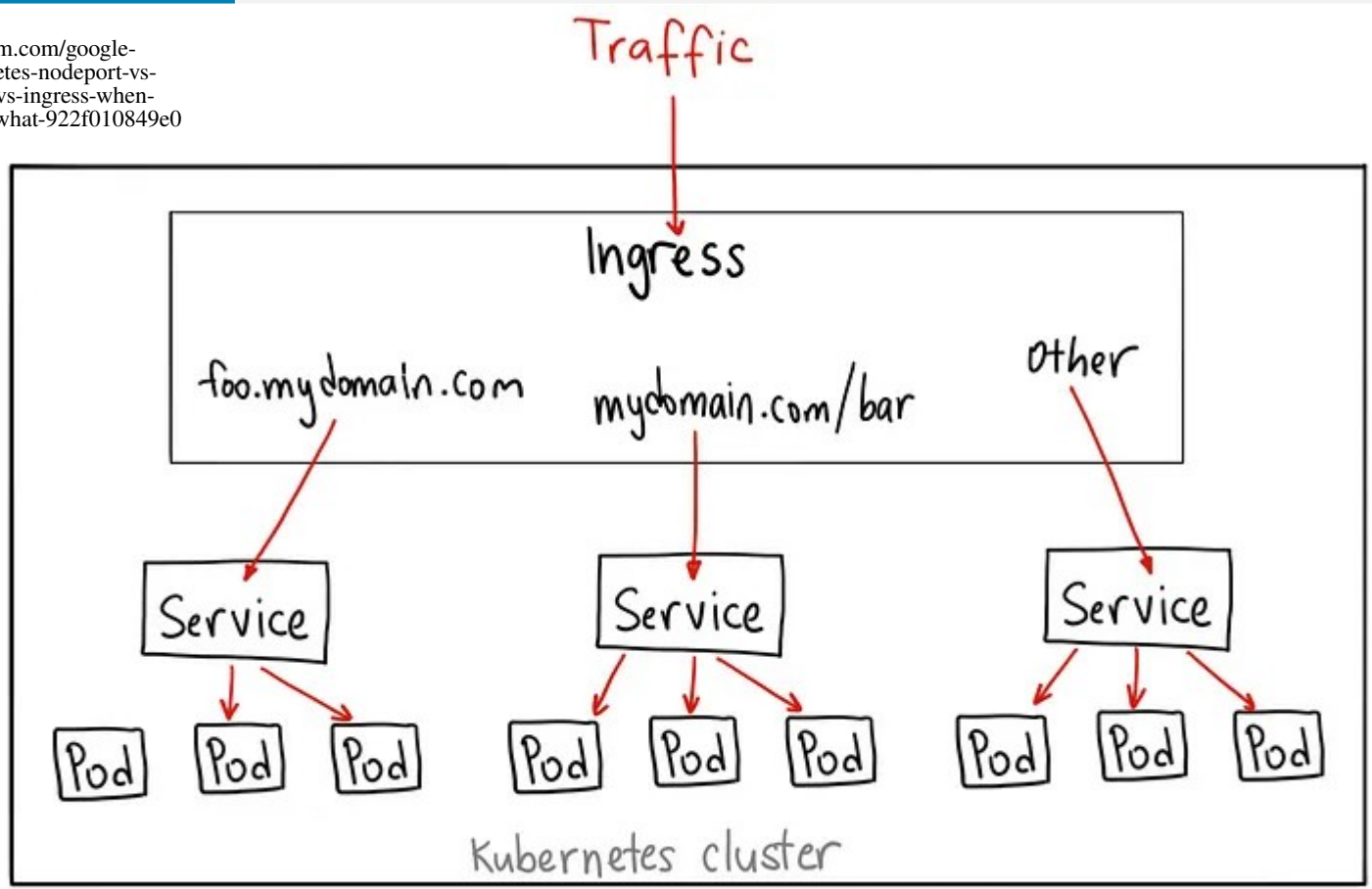


https://medium.com/google-cloud/understand-cloud-load-balancer-like-a-senior-engineer-d4f55f3111fc

L7 Lastverteilung

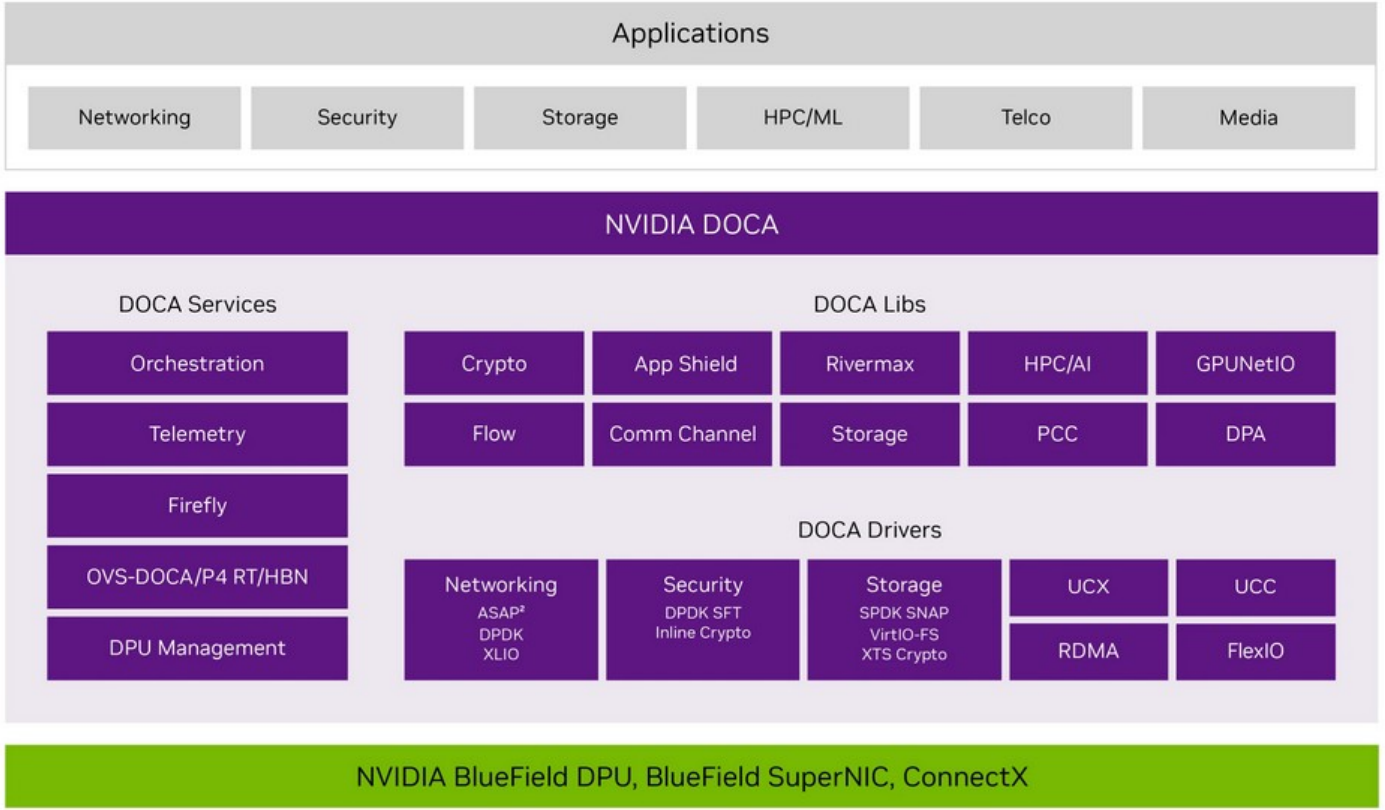
- Arbeitet auf OSI-Schicht 7
  → Anwendung
- Analysiert HTTP/S-Protokoll
  → Hostname, Pfade, Cookies
- Kubernetes Standart
  → Gateway API, Ingress (Deprecated)
- Vorteile:
  → perfekt für Webanwendungen und Microservices
  → TLS Terminierung möglich
- Nachteile:
  → deutlich höherer Ressourcenverbrauch durch parsing
  → schlechtere Leistung

https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0

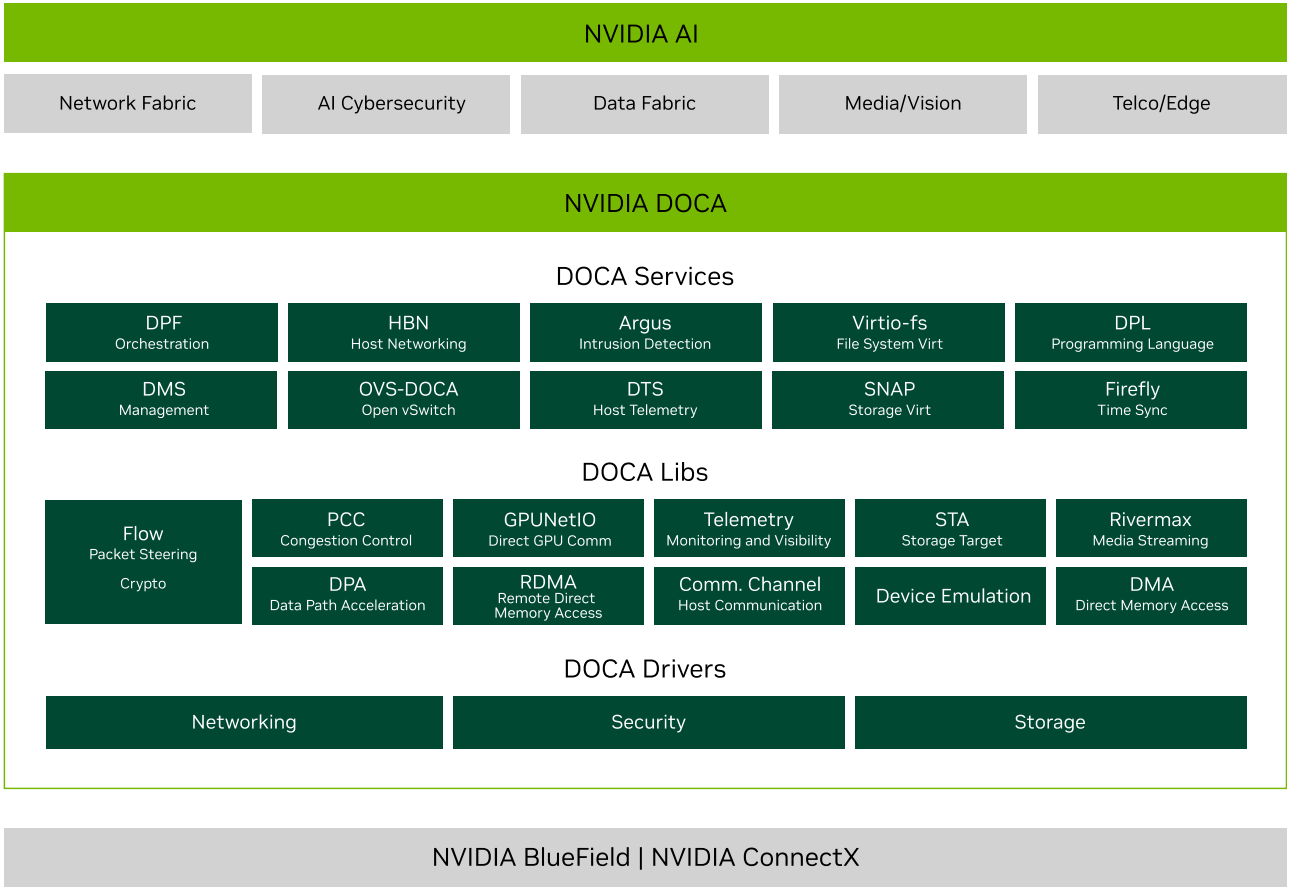


Datacenter On-a-Chip Architecture

- Vorranig SDK und Framework
- Soll primäre Anlaufpunkt sein wenn Zielarchitektur Bluefield ist
  → extrem breit aufgestellt und viele vorgefertigte Librarys
- Closed Source
  → kein Einblick hinter die Kulissen
- Schnelle Entwicklung im Fokus
  → „einfache“ Programmierung von Hardwarebeschleunigern
- Einfache Integration in Cloudnative Umgebungen
- Bildet Bibliotheken aber auch Treiber ab



https://developer.nvidia.com/networking/doca



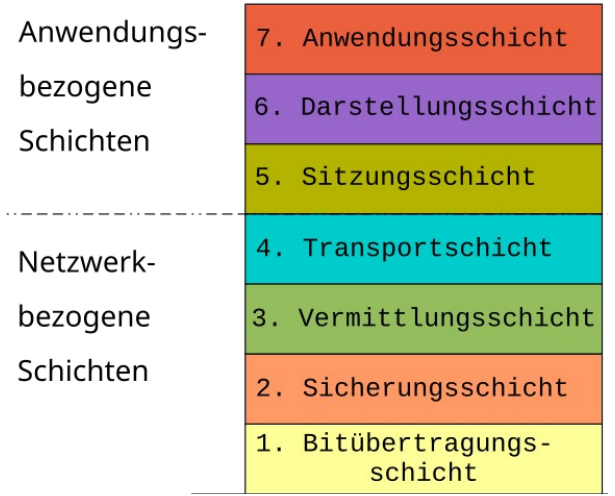
https://developer.nvidia.com/networking/doca

DOCA Flow

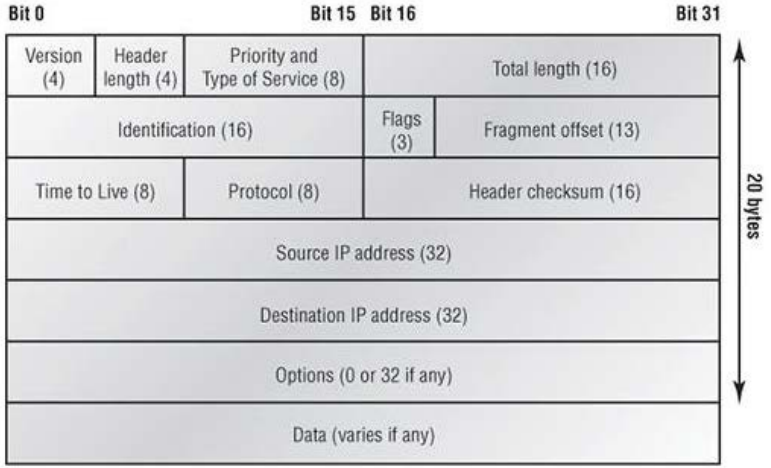
- Fundamentale API um generische Paketverarbeitungspipelines zu erstellen
- Es werden Pipes erstellt
  → bilden elementarste Einheit für Packet-Steering und Verarbeitung
  → bestehen aus Match, Monitoring, Actions und (Forward)
  → funktionale Einheiten sind Entries
- Pipes sollen vollständig auf Hardware laufen
  → nur Sonderfälle von Prozessor verarbeitet
- Verwendung von DPDK erlaubt auch Datenpaket konkret auszulesen
- Komplette Konfiguration deklarativ
  → obwohl als C Programm gestartet
- Menge an Beispielapplikationen gegeben

Matching

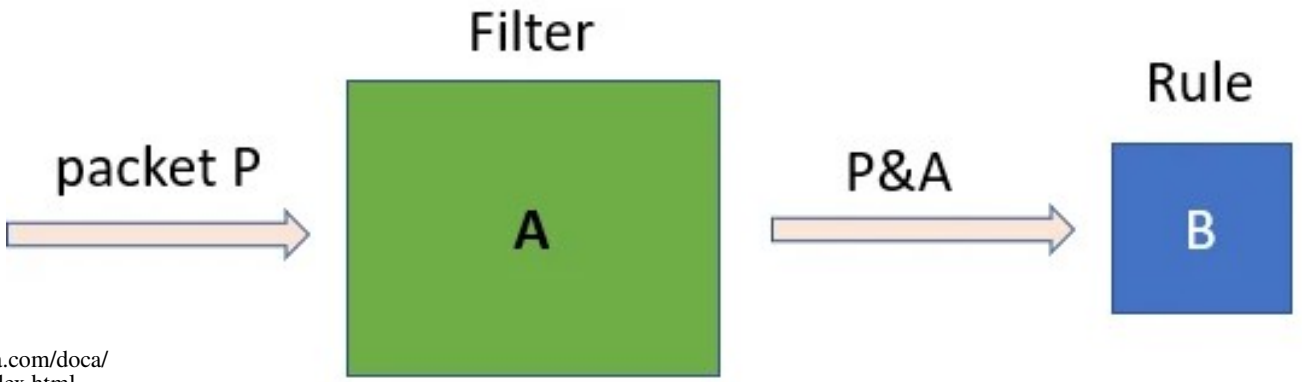
- Match kann auf bis zu zwei Layern stattfinden
- Erlaubte Match Layer sind u.a.:
  → MAC, VLAN, Ethertype
  → IPv4, IPv6
  → TCP, UDP, ICMP
- Wird mittels Bitfeld Operationen umgesetzt
  → dadurch gut durch Hardware zu beschleunigen
- Sowohl explizit als auch implizites Matching erlaubt
  → explizit match auf genaue Adressen
  → implizit match auf Adressklassen



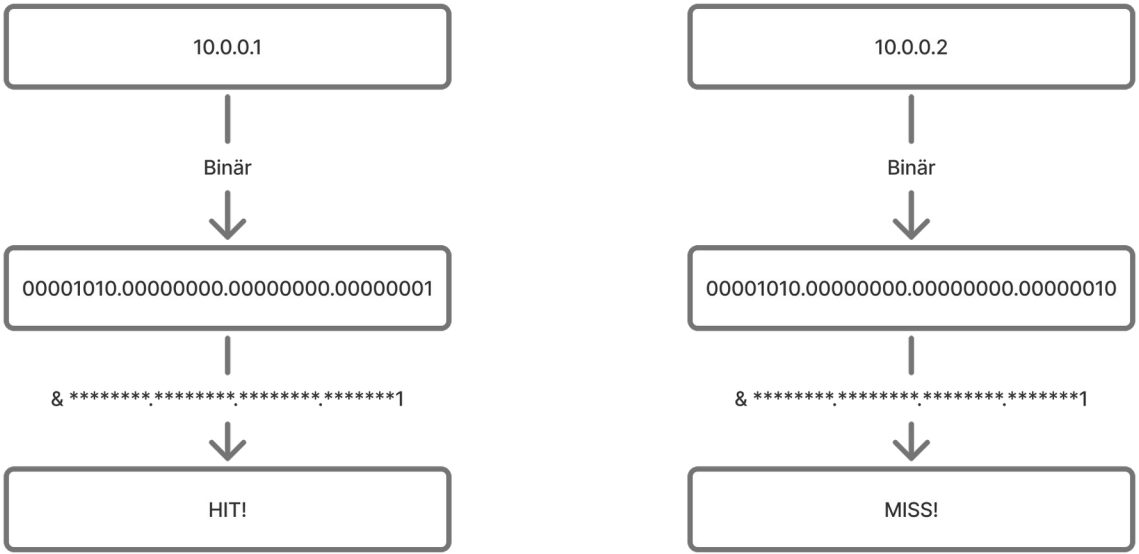
https://de.wikipedia.org/wiki/OSI-Modell#/media/Datei:ISO-OSI-7-Schichten-Modell_(in_Deutsch).svg



https://flylib.com/books/en/2.298.1.25/1/



https://docs.nvidia.com/doca/sdk/doca+flow/index.html



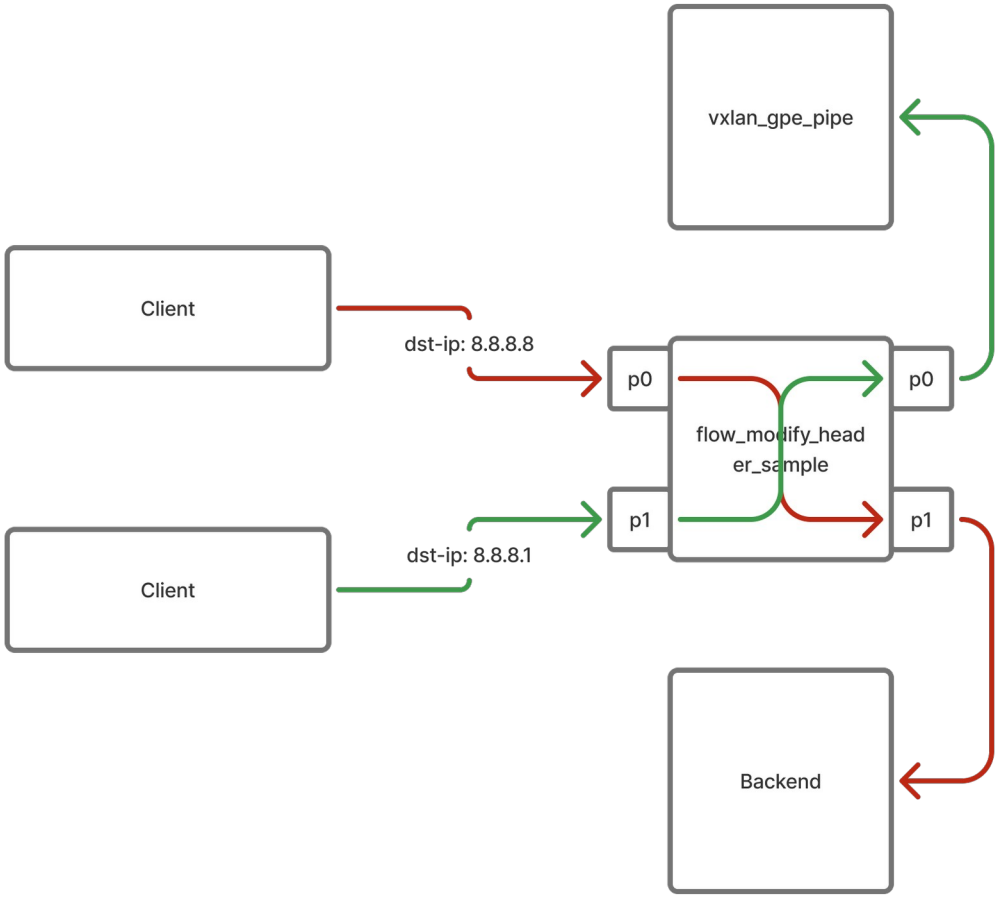
- Klassen der geraden und ungeraden IP-Adressen entstehen
  → 2 Klassen. Für mehr Klassen komplexere Filtermuster

Monitoring

- Datenfluss kann ohne Overhead untersucht werden
  → Auskunft über Pakete/Sekunde, Bytes/Sekunde
  → Bandbreite lässt sich errechnen
- Daten werden mittels intrinsischer Funktion in C-Struct geladen
  → Funktion aus der Library, kein genauer Einblick in die Funktionsweise
- Monitoring kann Pipes oder Entries zugeordnet werden
- Können statisch oder shared sein
  → shared erlaubt Zugriff auf selbe Monitorressource von unterschiedlichen Entries/Pipes

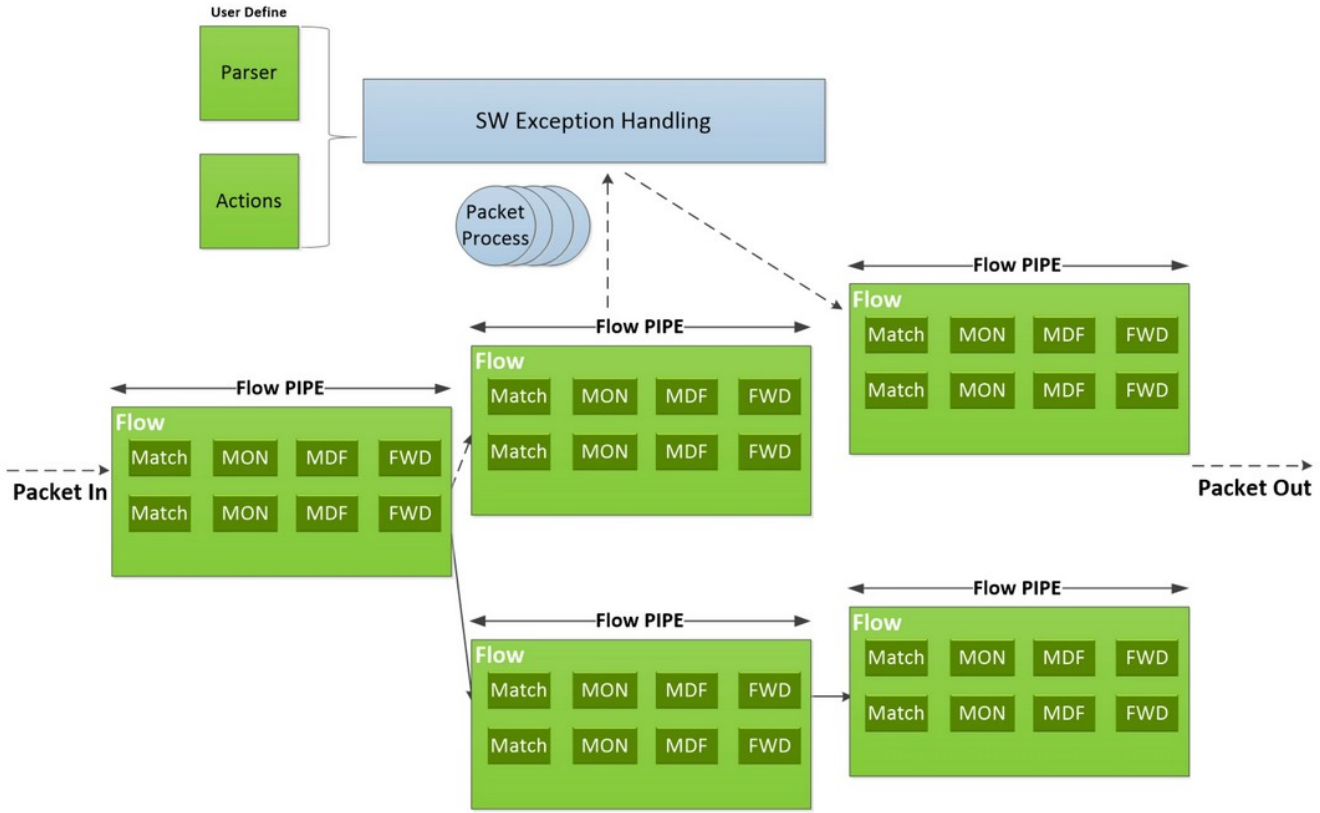
Actions

- Sind ausführende Komponente in der Pipe
  → sollen ebenfalls auf Hardware laufen, daher relativ begrenzt
- Erlaubte Actions sind:
  → ändern von MAC-Adressen (!)
  → ändern von IP-Adressen
  → ändern von L4-Ports
  → Tunnel
  → ändern von Metadaten
  → Encrypt/Decrypt
- Ebenfalls vollständige durch Bitfelder


vxlan_gpe_pipe
Client
dst-ip: 8.8.8.8
p0
p0
flow_modify_header_sample
p1
p1
Client
dst-ip: 8.8.8.1
Backend

Forward

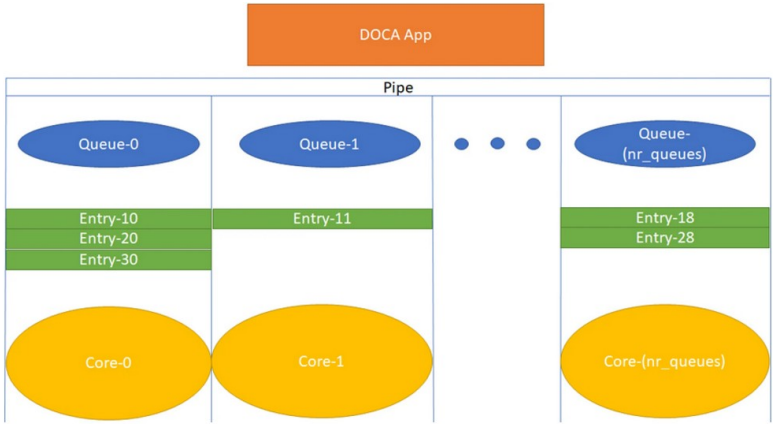
- Erlaubt Anbindungen von Pipes an nächste Einheiten
  → Kein direkter Schritt auf Hardwareeinheit
- Erlaubte Ziele sind:
  → Port
  → Pipe
  → Drop
  → (Software)
- Durch Verkettung mit anderer Pipe komplexere Anwendung möglich
  → Related Work misst oftmals wie schnell dynamische Änderungen möglich



https://docs.nvidia.com/doca/sdk/doca+flow/index.html

Entries

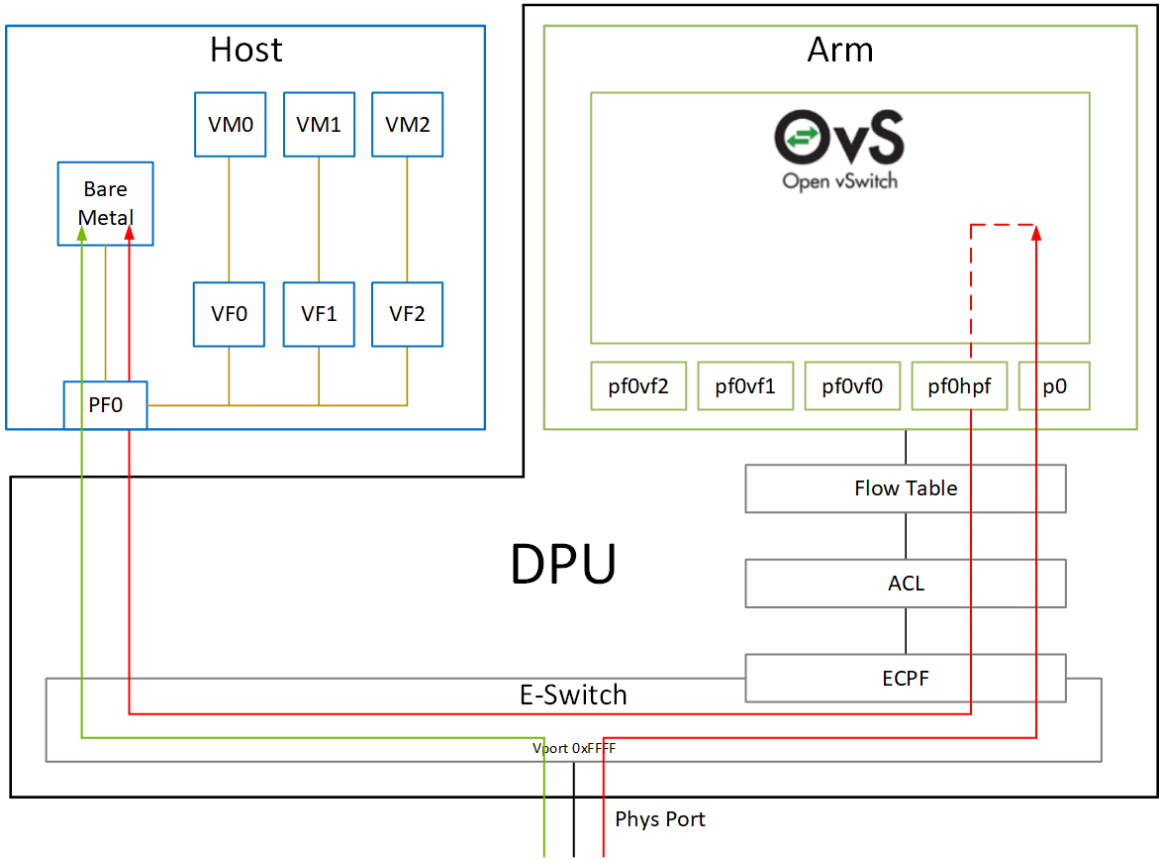
- Werden einer Pipe fest zugeordnet
- Sind die eigentlichen auf die Hardware ausgelagerten Funktionen
- Beliebig viele Entries pro Pipe erlaubt (bzw. keine Angabe)
- Können nur den Scope der Pipe erweitern
  → bsp. Können nicht auf anderen Port weiterleiten wenn nicht in Pipe angegeben



https://
docs.nvidia.com/doca/
sdk/doca+flow/
index.html

Open-vSwitch

- Software-basierter virtueller Switch
- Entwickelt für Virtualisierung & Cloud-Umgebungen
- Erlaubt u.a. Layer-2 u. Layer-3 Switching mit VLANs
- Anwendung in Hypervisor- und Container-Netzwerken
- Besteht aus:
  → Userspace-Daemon ovs-vswitchd
  → Kernen-Modul openvswitch.ko
  → CLI-Tools ovs-vsctl und ovs-ofctl
- DOCA-Open-vSwitch soll Regeln auf Hardware auslagern



https://docs.nvidia.com/doca/sdk/doca+flow/index.html

XenoFlow

- Eigentlicher LoadBalancer dieser Arbeit
- Baut vollständig auf DOCA Flow auf
  → einzige externe Abhängigkeit
- Läuft als DOCA Dienst vom Host aus
  → Kommunikation von Host an Hardware der BlueField bleibt Entwickler verborgen
- Liest Konfiguration der Backends mittels yaml-Datei
  → yaml typisches Format von Cloud-Umgebungen
- Trifft Lastverteilungsentscheidung aufgrund von Quell-IP-Adresse
  → somit L3-Lastverteiler
- Bisher nur statisches Lastverteilen möglich
- Quelloffen auf https://github.com/gespel/Bachelorarbeit

- Besteht aus einer Pipe mit jeweils einem Entry pro Backend
  → Ansatz mit mehreren Pipes wäre ebenfalls möglich
- Ingress auf Port-0 und Egress auf Port-1
- Verändert Ziel-MAC-Adresse sodass Backend Server erreicht wird

```
95      doca_flow_pipe_cfg_destroy(pipe_cfg);
96      return result;
97  }
98
99  static doca_error_t create_root_pipe(struct doca_flow_port *port,
100                                       int port_id,
101                                       enum doca_flow_l4_type_ext out_l4_type,
102                                       struct doca_flow_pipe **pipe)
103 {
104     struct doca_flow_match match;
105     struct doca_flow_match match_mask;
106     struct doca_flow_monitor monitor;
107     struct doca_flow_actions actions0, actions1, actions2, *actions_arr[2];
108     struct doca_flow_fwd fwd, fwd_miss;
109     struct doca_flow_pipe_cfg *pipe_cfg;
110     doca_error_t result;
111
112     memset(&match, 0, sizeof(match));
113     memset(&match_mask, 0, sizeof(match_mask));
114     memset(&monitor, 0, sizeof(monitor));
115     memset(&actions0, 0, sizeof(actions0));
116     memset(&actions1, 0, sizeof(actions1));
```

```
match.outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
match.outer.ip4.src_ip = BE_IPV4_ADDR(255, 255, 255, 255);
match_mask.outer.ip4.src_ip = BE_IPV4_ADDR(0, 0, 0, 1);
DOCA_LOG_INFO("%d", match_mask.outer.ip4.src_ip);
```

- Match-Maske um auf Klassen zu matchen
- Damit alle Bits verwendet werden müssen dieser zunächst auf 1 gesetzt werden
- Dann Aktivierung der relevanten Bits für Entry
- Noch kein konkreter Match → muss in Entries erfolgen

- Anlegen der Monitor Ressource
- ID-Bitfeld muss 1 gesetzt sein wenn ID in Entry festgelegt wird
- Typ muss shared sein
  → mehrere Entries greifen zu

```
monitor.counter_type = DOCA_FLOW_RESOURCE_TYPE_SHARED;
monitor.shared_counter.shared_counter_id = 0xffffffff;
```

```
SET_MAC_ADDR(actions0.outer.eth.dst_mac, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff);
SET_MAC_ADDR(actions0.outer.eth.src_mac, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff);

actions_arr[0] = &actions0;
```

- Action ebenfalls nur allgemein angelegt
  → konkreters Rewrite natürlich im Entry
- Mehrere Actions kann als Array übergeben werden

- Schlussendlich wird Forward festgelegt
- Miss muss separat deklariert werden
- Erstellung der konkrete Pipe mittels intrinsischer Funktion

```
fwd.type = DOCA_FLOW_FWD_PORT;
fwd.port_id = 1;
fwd_miss.type = DOCA_FLOW_FWD_DROP;

result = doca_flow_pipe_create(pipe_cfg, &fwd, &fwd_miss, pipe);
```

```
monitor.shared_counter.shared_counter_id = shared_counter_id;

match.outer.ip4.src_ip = BE_IPV4_ADDR(0, 0, 0, 1);

actions.action_idx = 0;

SET_MAC_ADDR(actions.outer.eth.dst_mac, 0xe8, 0xeb, 0xd3, 0x9c, 0x71, 0xac);
SET_MAC_ADDR(actions.outer.eth.src_mac, 0xc4, 0x70, 0xbd, 0xa0, 0x56, 0xbd);

result = doca_flow_pipe_add_entry(0, pipe, &match, &actions, &monitor, NULL, 0, status, &entry_mac);
```

- Im eigentlichen Entry nur noch einfügen der Werte
- Übergabe des Entries an Pipe mittels intrinsischer Funktion die Referenz zur Pipe Datenstruktur erhält

Messungen und Experimente

- Zu Beginn der Arbeit wurden 3 Forschungsfragen formuliert
  → Ziel war es den Lastverteiler auf der gegebenen Hardware gründlich zu testen
- Es wurden 3 unterschiedliche Experimente durchgeführt
  → Hardwareauslagerung erst nach OvS Konfiguration

  → Ist das Versprechen von Zero-Overhead-Verarbeitung ohne Paketverlust aus der NVidia Dokumentation so wahrheitsgetreu?
  → Ist es tatsächlich möglich, auch unabhängig von der Paketgröße, immer die angegebenen 400 Gbit/s zu verarbeiten?
  → Inwiefern beeinflusst die Last des Load-Balancers die Round-Trip-Time, also somit die Latenz?

Hardware

- In Experimenten wurden 5 Knoten verwendet
  - → fips1 - Stellt den Testclient dar, der im folgenden UDP-DNS-Requests an den Server sendet
  - → fips2 - Backend Server auf dem sowohl Grundlast ankommt als auch der DNS-Server läuft
  - → fips3 - Backend Server, auf dem nur Grundlast ankommt
  - → fips4 - Lasterzeugender Server
  - → fips5 - XenoFlow Knoten, in dem die BlueField-3 verbaut ist

- fips-* Knoten

  → NVIDIA Mellanox MCX623106AN-CDAT ConnectX®-6 Dx EN Netzwerkkarte, 100GbE

  → 32 Kerne u. 64 Threads Intel Xeon Silver 4514Y mit 128 GB DDR4 RAM

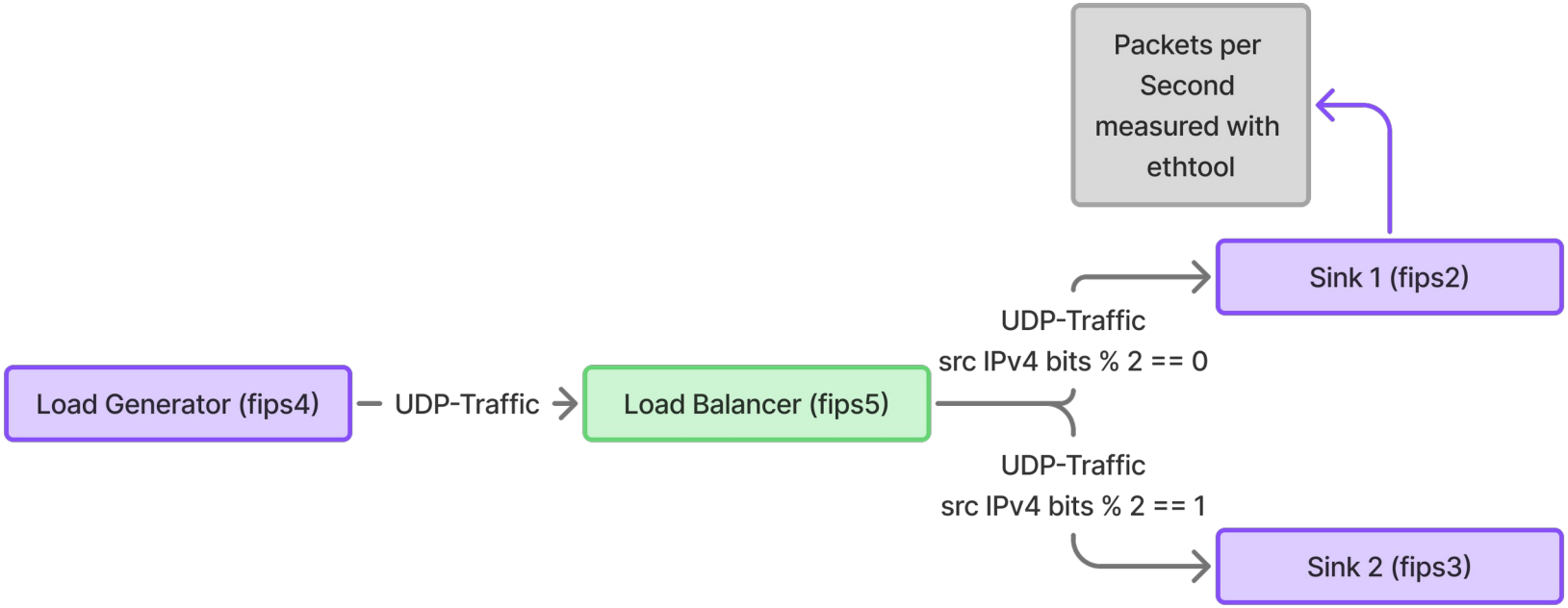
- BlueField-3

  → ARM-Cortex-A78AE 16 Kerne mit 32 GB DDR5 RAM

- Co-Prozessor (Data Path Accelerator) RISC-V Architektur Basistakt von 1.8 GHz.

Experiment 1: Verarbeitucngsgeschwindigkeit

- Wie viele Pakete pro Sekunde kann die Bluefield maximal verarbeiten?
- Lasterzeuger schickt bestimmte Anzahl von Paketen pro Sekunde an XenoFlow + Bluefield
  → Lastverteiler leitet an ein Backend weiter
  → auf Backend werden die ankommenden Pakete mittels ethtool gemessen
  → müssen auf Hardware auslesen, da Linux Kernel schon weit vorher in die Knie geht
- Für Lasterzeugung wird T-Rex verwendet
  → schickt direkt mittels DPDK
  → Kernel to Userspace


Packets per Second measured with ethtool
Sink 1 (fips2)
UDP-Traffic
src IPv4 bits % 2 == 0
Load Generator (fips4)
UDP-Traffic
Load Balancer (fips5)
UDP-Traffic
src IPv4 bits % 2 == 1
Sink 2 (fips3)

Experiment 2: Latenzänderung

- Wie verändert sich die Latenz wenn sich die Last auf die Bluefield/Lastverteiler verändert
- Lasterzeuger schickt bestimmte Anzahl an Paketen pro Sekunde Grundlast an Lastverteiler

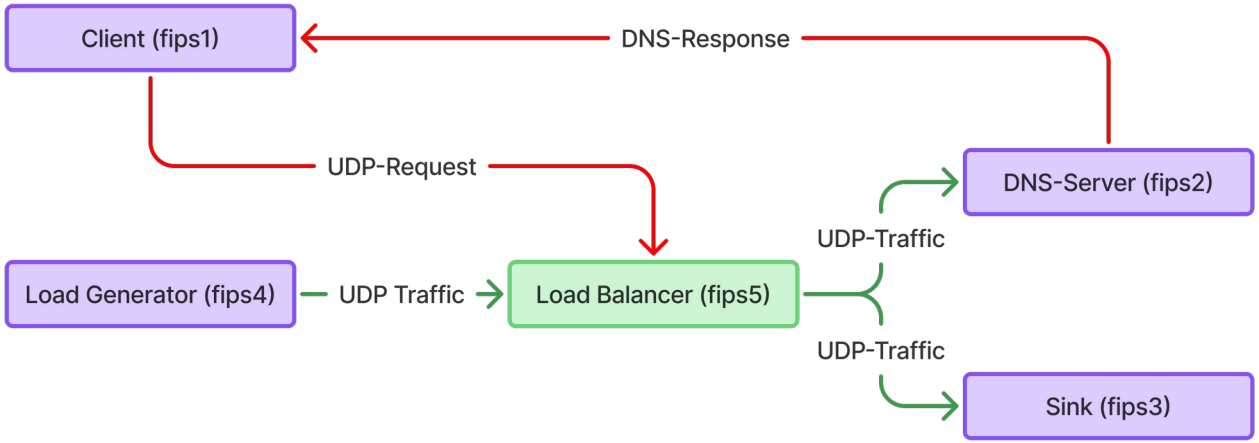
  → Verteilt Grundlast auf zwei Backends mit unterschiedlichen Verhältnissen

  → 0/100, 50/50, 70/30, 100/1
- Auf einem Backend läuft DNS-Server

  → bekommt Test-Requests von DNS-Client

  → gemessen wird die Round-Trip-Time (RTT) vom DNS-Request

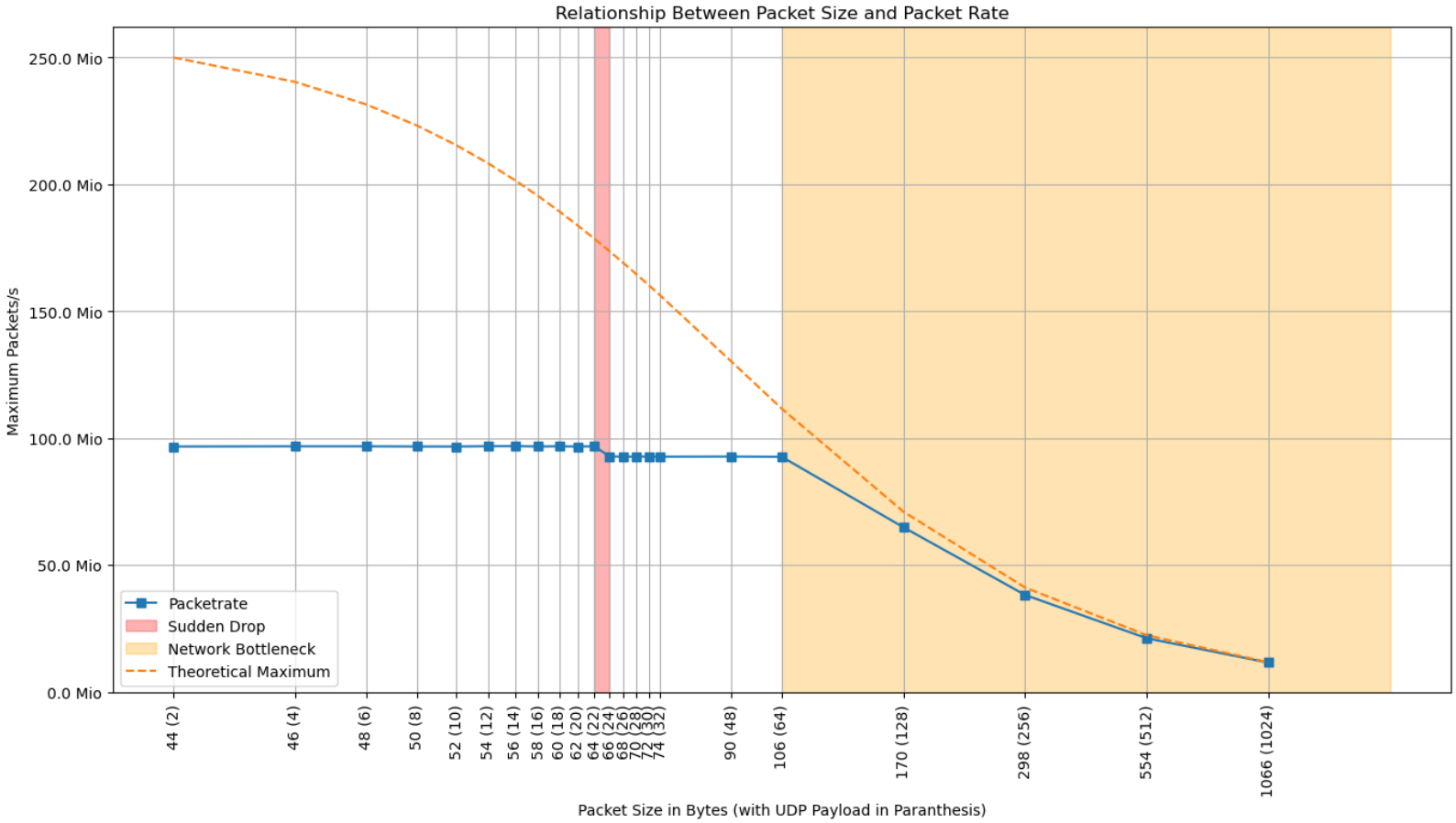
```
07:09:08.767952 a0:88:c2:b6:14:1a > c4:70:bd:a0:56:ac, ethertype IPv4 (0x0800), length 71: 10.3.10.42.44444 > 10.3.10.45.53: 4660+ A? example.org. (29)
07:09:08.769756 a0:88:c2:b5:f4:5a > a0:88:c2:b6:14:1a, ethertype IPv4 (0x0800), length 98: 10.3.10.45.53 > 10.3.10.43.5353: 4660*- 1/0/0 A 1.1.1.1 (56)
07:09:09.064001 a0:88:c2:b6:14:1a > c4:70:bd:a0:56:ac, ethertype IPv4 (0x0800), length 70: 10.3.10.42.44444 > 10.3.10.45.53: 4660+ A? test.local. (28)
07:09:09.065816 a0:88:c2:b5:f4:5a > a0:88:c2:b6:14:1a, ethertype IPv4 (0x0800), length 96: 10.3.10.45.53 > 10.3.10.43.5353: 4660*- 1/0/0 A 1.1.1.1 (54)
07:09:09.359996 a0:88:c2:b6:14:1a > c4:70:bd:a0:56:ac, ethertype IPv4 (0x0800), length 70: 10.3.10.42.44444 > 10.3.10.45.53: 4660+ A? test.local. (28)
07:09:09.361817 a0:88:c2:b5:f4:5a > a0:88:c2:b6:14:1a, ethertype IPv4 (0x0800), length 96: 10.3.10.45.53 > 10.3.10.43.5353: 4660*- 1/0/0 A 1.1.1.1 (54)
07:09:09.652000 a0:88:c2:b6:14:1a > c4:70:bd:a0:56:ac, ethertype IPv4 (0x0800), length 71: 10.3.10.42.44444 > 10.3.10.45.53: 4660+ A? example.org. (29)
07:09:09.653865 a0:88:c2:b5:f4:5a > a0:88:c2:b6:14:1a, ethertype IPv4 (0x0800), length 98: 10.3.10.45.53 > 10.3.10.43.5353: 4660*- 1/0/0 A 1.1.1.1 (56)
```

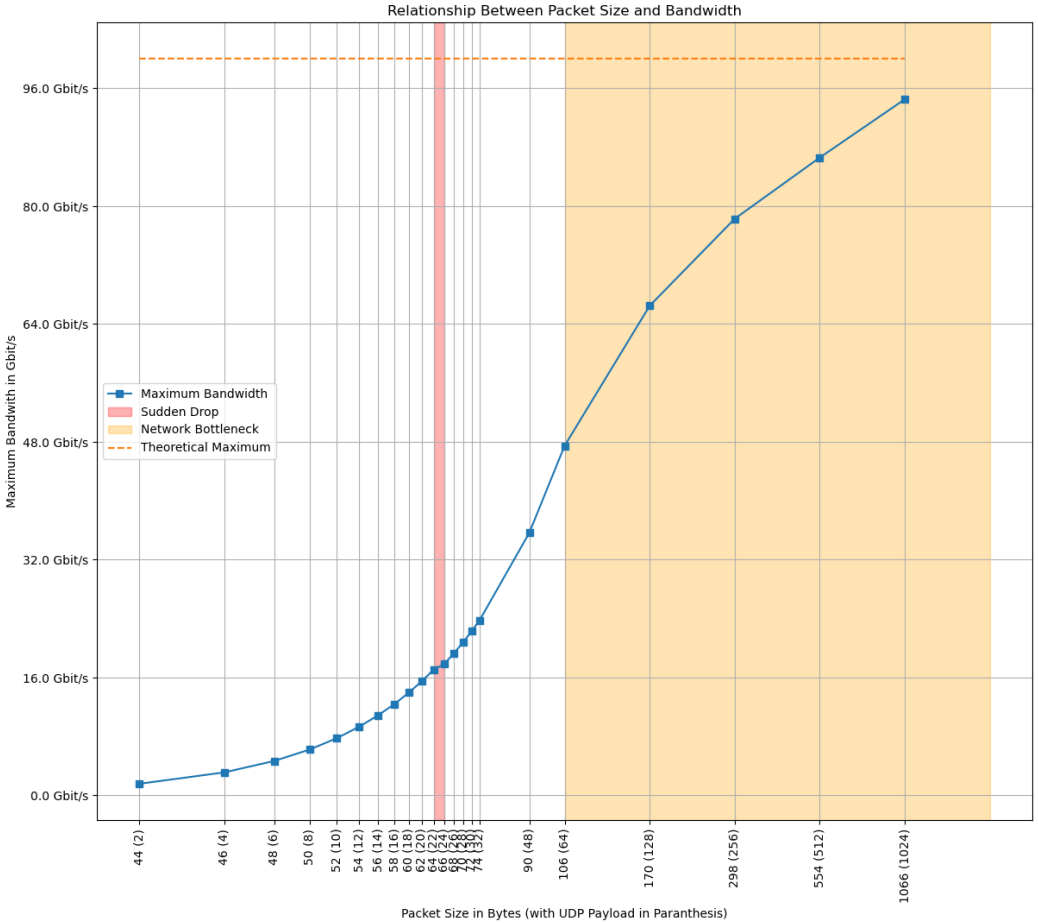


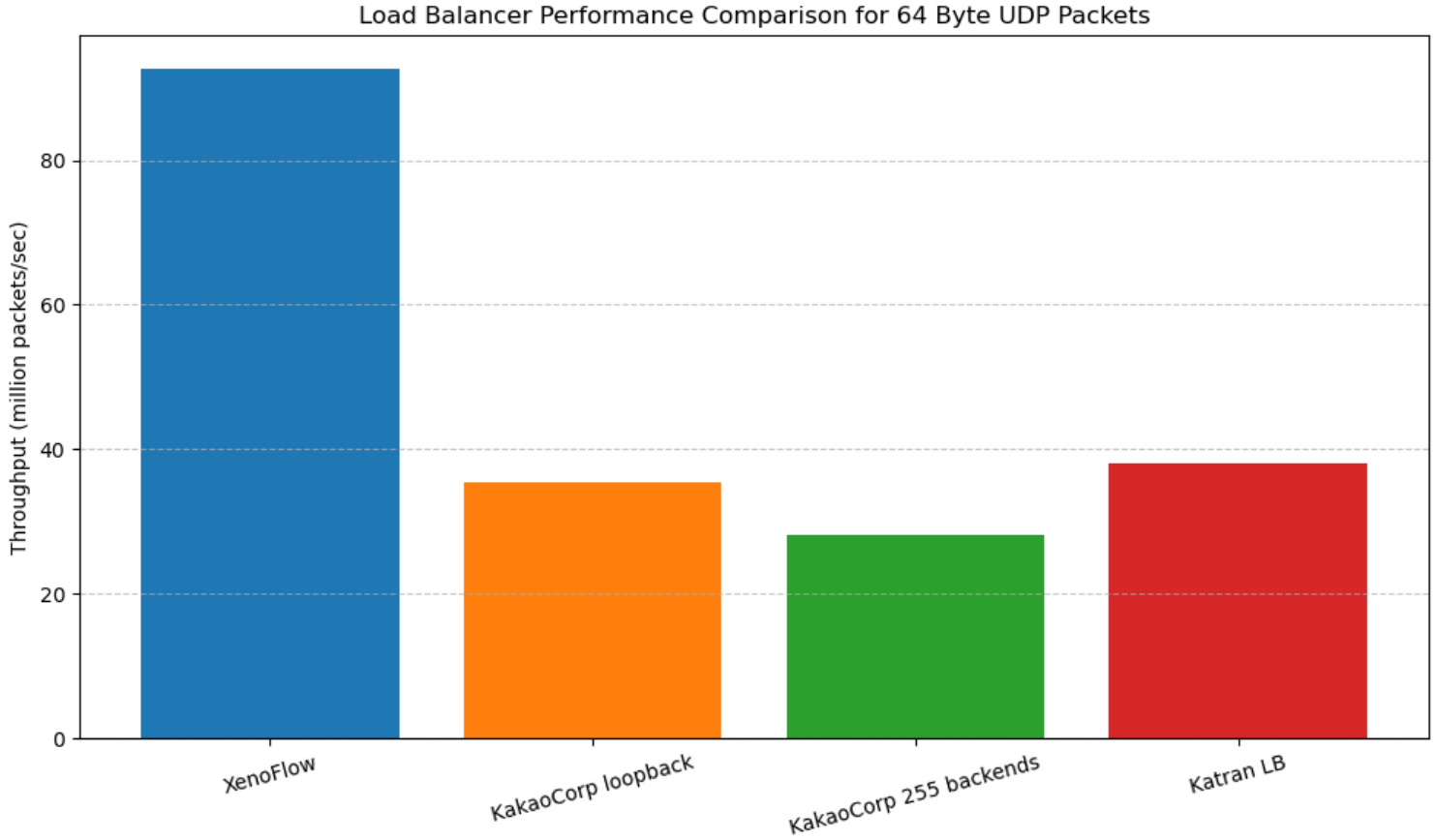
Experiment 3: Paketverlust

- Wie groß ist der Anteil der verlorenen Pakete bei einer bestimmten Grundlast
- Lasterzeuger erzeugt Grundlast auf Lastverteiler
- Client sendet DNS-Requests die an DNS-Server-Backend weitergeleitet werden
- Gemessen wird die Menge der DNS-Anfragen des Clients die verloren gehen und somit nicht beantwortet werden

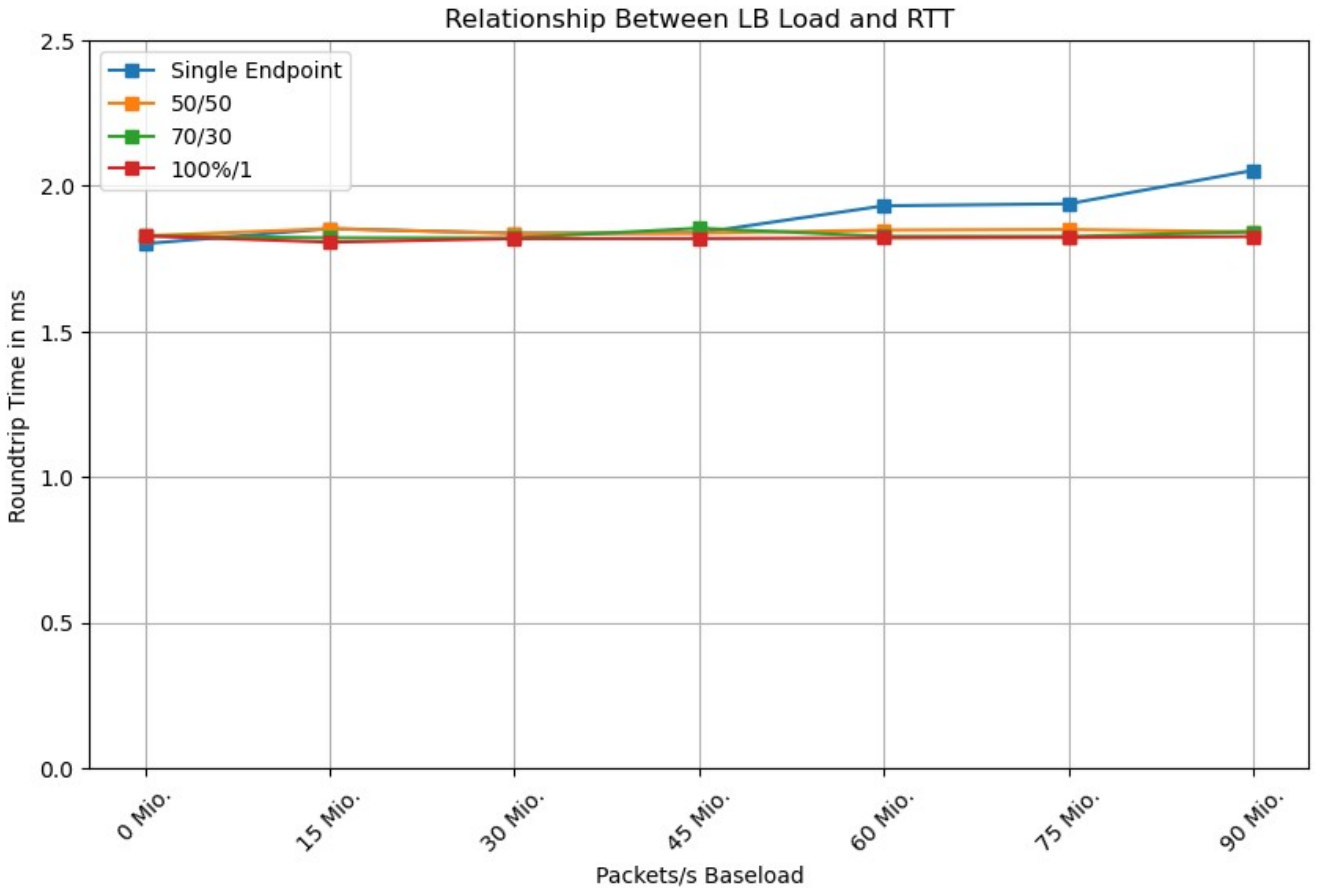
Ergebnisse!

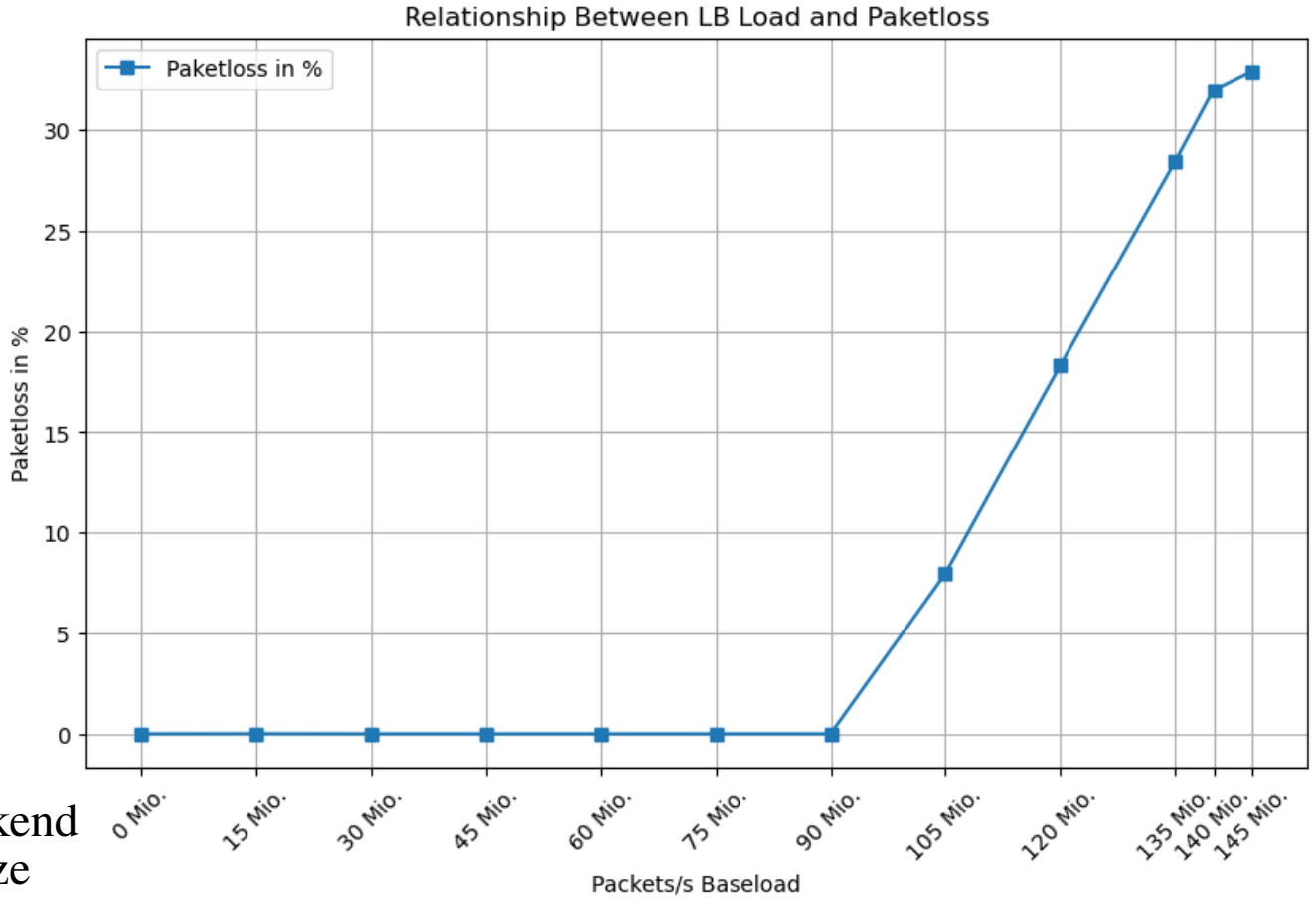

Relationship Between Packet Size and Packet Rate
Maximum Packets/s
250.0 Mio
200.0 Mio
150.0 Mio
100.0 Mio
50.0 Mio
0.0 Mio
Packetrate
Sudden Drop
Network Bottleneck
Theoretical Maximum
44 (2)
46 (4)
48 (6)
50 (8)
52 (10)
54 (12)
56 (14)
58 (16)
60 (18)
62 (20)
64 (22)
66 (24)
68 (26)
70 (28)
72 (30)
74 (32)
90 (48)
106 (64)
170 (128)
298 (256)
554 (512)
1066 (1024)
Packet Size in Bytes (with UDP Payload in Paranthesis)


Relationship Between Packet Size and Bandwidth
Maximum Bandwidth in Gbit/s
96.0 Gbit/s
80.0 Gbit/s
64.0 Gbit/s
48.0 Gbit/s
32.0 Gbit/s
16.0 Gbit/s
0.0 Gbit/s
Maximum Bandwidth
Sudden Drop
Network Bottleneck
Theoretical Maximum
44 (2)
46 (4)
48 (6)
50 (8)
52 (10)
54 (12)
56 (14)
58 (16)
60 (18)
62 (20)
64 (22)
66 (24)
68 (26)
70 (28)
72 (30)
74 (32)
90 (48)
106 (64)
170 (128)
298 (256)
554 (512)
1066 (1024)
Packet Size in Bytes (with UDP Payload in Paranthesis)


Load Balancer Performance Comparison for 64 Byte UDP Packets
Throughput (million packets/sec)
0
20
40
60
80
XenoFlow
KakaoCorp loopback
KakaoCorp 255 backends
Katran LB

Basislatenz:
1.82 ms





Single-Backend
64 Bytes size

Auswertung

- Ist das Versprechen von Zero-Overhead-Verarbeitung ohne Paketverlust aus der NVidia Dokumentation so wahrheitsgetreu?

  → Nein, es ist definitiv Paketverlust messbar (auch unter 400 Gbit/s)
- Ist es tatsächlich möglich, auch unabhängig von der Paketgröße, immer die angegebenen 400 Gbit/s zu verarbeiten?

  → Nein, nur große Pakete erreichen überhaupt große Bandbreiten.
- Inwiefern beeinflusst die Last des Load-Balancers die Round-Trip-Time, also somit die Latenz?

  → Es konnte kein Zusammenhang zwischen Last auf Lastverteiler/Bluefield und Latenz gemessen werden. Somit wahr.

Fazit

- SmartNICs in diesem Beispiel durchaus geeignet für Anwendungen
- Implementierung muss sich nach DOCA Flow Framework richten

  → sonst unter Umständen nicht hardwarebeschleunigt
- Marketing bleibt Marketing

  → viele Claims sicherlich nicht haltbar

  → bewusste Auslassungen
- Loadbalancer aktuell Proof-of-Concept

Ausblick

- Ausbau des Lastverteilers

  → dynamische Anpassung der Backends

  → „Alive“ Probing
- Implementierung von yaml-Parser für DOCA Konfiguration
- Weitere Anwendungen bzgl. Hardwareauslagerung

  → meisten Anwendungen sind Closed-Source und Leben irgendwo bei Google, Azure, AWS etc.
- Auslagerung von ML-Anwendungen mittels DMA oder BluefieldX

  → Stichwort Edge-Computing

Links und Referenzen

Repository

Bachelorarbeit und Referenzen

Vielen Dank